The ITOS sockets API

providing robust sockets for ITOS applications

This document describes a set of network socket handling functions provided in the main ITOS library. These functions provide consistent socket handling and capabilities across ITOS applications and feature TCP wrappers support on server-side TCP sockets and automatic retry and failover support on client-side TCP sockets.

1 Initialization

The ITOS sockets library operates using a `struct socket_instance`, which contains everything you and the library needs to know about a socket. This chapter discusses the structure and the convenience functions for initializing it.

1.1 socket_instance

The `struct socket_instance` contains the following members:

`char *app_name`
: The name of the application is used for two purposes: For server-side sockets, the library calls TCP Wrappers for permission to accept the incoming connection request, and this is the name Wrappers will look for in the '`/etc/hosts.allow`' (or '`/etc/hosts.deny`' file. It also is used when printing error, warning, and informational messages.

`enum socket_transport txport`
: The IP protocol and, for TCP, the role to play in forming a connection. One of:

  `ST_TCP_CLIENT`
  : for active (client-side) TCP connections,

  `ST_TCP_SERVER`
  : for passive (server-side) TCP connections,

  `ST_UDP`
  : for UDP messaging, and

  `ST_NULL`
  : to indicate that the transport type is unset.

`struct socket_host_port local`
: The local host name, IP address, and port number associated with this socket. These values are set by the library, never by the user. [The probably should be settable by the user in the case of a computer with multiple IP addresses (either multiple interfaces or multiple addresses on a single interface), but not been an issue so far.]

`struct socket_host_port foreign`
: The foreign host name, IP address, and port number associated with this socket. These values are set by the library, never by the user. (The socket_set_foreign_server() and socket_tcp_accept() set this.)

`struct socket_host_port listen`
: The local host name, IP address, and port number on which we are listening for a TCP connection request from a foreign host. The port number is set by the user, usually by calling `socket_server_instance()`. The host is ignored, though it should be used in case the host has multiple interfaces.

`int data_fd`
: The file descriptor for the connected socket. This value may be obtained from the structure using `socket_data_fd()`.

`int listen_fd`
: The file descriptor for the listen socket, if this `socket_instance` is a TCP server socket. This value may be obtained from the structure using `socket_listen_fd()`.

`continue_listening`
: If non-zero, do not close the listen socket after accepting a connection. Set this with `socket_server_instance()`.

`listen_timeout`
: If this `socket_instance` is a TCP server socket, this is the number of seconds to wait for a peer to connect to the listen socket. If zero, wait forever. If a peer fails to connect within the timeout, the socket is closed and the `timed_out` flag is set.

`int timed_out`
: If this `socket_instance` is a TCP server socket, this is set non-zero by the library if a peer does not connect to the server socket within `listen_timeout` seconds.

`int rejected`
: If this `socket_instance` is a TCP server socket, this is set non-zero by the library if TCP Wrappers rejects a connection request.

`struct socket_server_set server_set`
: If this `socket_instance` is a TCP client socket, this is the intended server and it's alternates. See below for a full explanation. This substructure is set by the user through calls to `socket_client_instance()`, `socket_client_failover()`, and associated functions.

`void (*errfn)(char *, ...)`
: This is a pointer to a function which the library will call if it encounters an error. Messages include the `errno` value and associated error string, where appropriate.

`void (*warnfn)(char *, ...)`
: This is a pointer to a function which the library will call if it needs to issue a warning. Warnings are issued when TCP Wrappers rejects a connection request and when calls to `getaddrinfo()` or `getnameinfo()` fail.

`void (*msgfn)(char *, ...)`
: This is a pointer to a function which the library will call to announce a change of state. It is called when a listen port is established, and when a connection is established.

The `struct socket_host_port` contains the following members:

`char host[NI_MAXHOST]`
: The host name associated with the socket. Every computer has at least two: one for the loopback and a real network interface, but some will have multiple real network interfaces. This can be set to a host name or IP address string

in IPv4 or IPv6 representation. The library will set it to the cannonical host name.

`char ipaddr[INET6_ADDRSTRLEN]`
: The IP address associated with the host, as a string. This is set by the library, never by the user!

`char port[NI_MAXSERV]`
: The port number or service name (from '`/etc/services`' or a name service) associated with the socket.

The `struct socket_server_set` is used for TCP sockets playing the client (active) role. Its primary feature is an array of server hosts to which the library will attempt to connect the socket. The first entry in the array must be populated and is the primary server. The other entries are alternates to which the application may fail over if the primary server cannot be contacted.

Counters are maintained internally to the structure relative to `max_server_retries` and `max_server_cycles`. When a connection is successfully established, those counters are reset. If the connection is broken and the application tries to re-establish a connection, it can retry and cycle through alternates for the full maximums.

`struct socket_host_port servers[SERVER_SET_SIZE]`
: The set of servers to which the socket may be connected. They are contacted in order, so the primary server is `servers[0]`, which is set by `socket_client_instance()`. The other elements in the array are populated by `socket_client_failover()`.

`int max_server_retries`
: The maximum number of times to attempt to connect to a server before trying the next server in the list.

`int max_server_cycles`
: The maximum number of times to cycle through the server list if none of the hosts can be contacted.

`float server_retry_interval`
: The time to wait between retries of a given server, or failover to the next server.

1.2 socket_instance_init

`void socket_instance_init(struct socket_instance *`*sock*`, char *`*app_name*`, char *`*sock_name*`)`

This function should be called by to initialize the `socket_instance` structure. It sets the application name to *app_name* and the socket instance name to *sock_name*. It also sets the file descriptors to `-1`, `server_set.max_server_cycles` to `1`, `server_set.server_retry_interval` to `2.0` seconds, and all other values to zero.

1.3 socket_set_msg_fns

```
void socket_set_msg_fns(struct socket_instance *sock,
                        void (*errfn)(char *, ...),
                        void (*warnfn)(char *, ...),
                        void (*msgfn)(char *, ...))
```

This function should be used to set the message function pointers in *sock*. The function pointed to by *errfn* is called on errors which prevent the library from completing its task; *warnfn* is called on errors which do not prevent the library from doing its job; and *msgfn* is called to announce the creation of a listen socket or the formation of a connection.

The message formats have a consistent structure. Error and warning messages begin with the application name from `sock->app_name` followed by the library function name in parenthesis, a colon, the system call name with any relevant call information, another colon, the parenthesized `errno` value, and the associated error string.

Informational messages also begin with the application name, and then give the action performed and the host name, IP address, and port number for both ends of the connection if it is reporting a connection formation. (Messages reporting that a listen socket has been created only contain information about the local end of the socket since the other end is not connected.)

1.4 socket_instance_setup

```
int socket_instance_setup(struct socket_instance *sock, int *argc,
                          char **argv, int posix_behavior);
```

This function takes command line arguments and returns an initialized socket instance structure, ready to be passed to socket_setup() or an allied function. Upon return, items used by `socket_instance_setup()` have been removed from *argv[]*, and *argc* has been decremented accordingly.

If *posix_behavior* is non-zero, processing of *argv[]* will stop with the first unrecognized option. Otherwise, processing will continue until the end of *argv[]* is reached. *arg_count*, if not `NULL`, is set to the number of arguments consumed.

(The `getopt_long()`, function is used to parse the command-line options. If *posix_behavior* is 0, `"-"` is passed as `getopt_long()`'s third argument; otherwise, `"+"` is passed. The latter causes POSIX behavior – option processing stops with the first unrecognized option – while the former allows options not recognized by `socket_instance_setup()` to be mixed with those which are recognized, but prevents permutation of the unrecognized options.)

The function returns non-zero on errors.

The following are recognized:

'`--socket`'
: does two things: Allows you to assign a name to this socket instance, and it ends option scanning like the '`--break`' option.

'`--txport`'
: may be one of:

- ‘`client_tcp`’
- ‘`server_tcp`’
- ‘`udp`’

‘`--af`’ the address family; one of:

‘`inet`’	for IPV4
‘`inet6`’	for IPV6

‘`--host`’ shorthand for ‘`--foreign-host`’.

‘`--port`’ shorthand for ‘`--foreign-port`’.

‘`--foreign-host`’
required only for the ‘`client_tcp`’ transport and for the ‘`udp`’ transport when data is being transmitted from the application over the UDP socket. This is the host to which to initiate a TCP connection, or to which to send UDP datagrams.

‘`--foreign-port`’
required only for the ‘`client_tcp`’ transport and for the ‘`udp`’ transport when data is being transmitted from the application over the UDP socket. This is the port on the foreign host to which to initiate a TCP connection, or to which to send UDP datagrams.

‘`--local-host`’
for systems with multiple network interfaces, this may be used to specify on which interface a socket should operate.

‘`--local-port`’
required only for ‘`server_tcp`’ transport, and for ‘`udp`’ transport when data is being received by the application over the UDP socket.

‘`--max-cycles`’
the maximum number of times to repeat the cycle through the failover list for ‘`client_tcp`’ connections.

‘`--max-retries`’
the maximum number of times to retry a ‘`client_tcp`’ connection which fails to connect.

‘`--retry-interval`’
the number of seconds between successive ‘`client_tcp`’ connection attempts.

‘`--listen-timeout`’
the number of seconds for which the program should present a ‘`server_tcp`’ listen socket.

‘`--keep-listening`’
if given, maintain the ‘`server_tcp`’ listen socket after connections are accepted.

‘`--stop-listening`’
if given, close the ‘`server_tcp`’ listben socket after a connection is accepted.

‘`--break`’ if given, stops option scanning as if the end of the option list had been reached.

1.5 socket_list_setup

```
struct socket_instance *
socket_list_setup(int *error, socket_instance *template,
                  int *argc, char **argv, int posix_behavior);
```

This function takes command line arguments and returns an initialized linked list of socket instance structures by calling `socket_instance_setup()`, repeatedly until it stops consuming arguments or encounters an error. The *argc*, *argv[]*, and *posix_behavior* arguments are handled as for `socket_instance_setup()`.

The *template* argument should be initialized with `socket_instance_init()` and `socket_set_msg_fns()` before being passed to this function. The application name and message functions from the template are copied into each new socket instance created. Also, the message functions may be called from within `socket_list_setup()` and `socket_instance_setup()`.

On errors, the variable pointed to by *error* will be non-zero. If no complete `socket_instance` could be formed from the command line arguments, a `NULL` is returned.

For each `socket_instance` in the list, pass the structure pointer to `socket_next_instance()` to get the next structure in the list. `NULL` is returned when the end of the list has been reached.

1.6 socket_server_instance

```
void socket_server_instance(struct socket_instance *sock, int port,
                            char *host, int continue_listening)
```

This function sets up *sock* to be a TCP server side (passive or listening) socket. The application will listen on the *port* for a TCP connection. The *host* argument currently is unused, but probably should be used to better handle computers with multiple addresses.

Normally, after accepting a connection, the library will close the listen socket, but if *continue_listening* is non-zero, the listen port will be left open. This allows applications to accept multiple connections.

1.7 socket_client_instance

```
void socket_client_instance(struct socket_instance *sock,
                            char *host, int port)
```

This function sets up *sock* to be a TCP client side (active) socket. The *host* and *port* are used to populate *sock*`->server_set.servers[0]` – the primary server to which the socket should be connected. The *host* argument may be a host name, an IPv4 address in dotted-decimal notation, or an IPv6 address in hex string; the *port* can be a numeric port number (in ASCII) or a service name from '`/etc/services`' (or the equivalent NIS or NIS+ map).

1.8 socket_client_failover

```
void socket_client_failover(struct socket_instance *sock,
                            char *host, int port)
```

This function adds alternate servers to *sock*, which should already have been set up to be TCP client socket by calling `socket_client_instance()`. The *host* and *port* are used to populate *sock*`->server_set.servers` entries beyond entry zero.

1.9 socket_udp_instance

```
void socket_udp_instance(struct socket_instance *sock,
                         char *lhost, int lport,
                         char *fhost, int fport)
```

This function sets up *sock* to be a UDP socket. The *lport* is the local port which should be assigned to the socket. The *fhost* and *fport* are the foreign host and port associated with the socket, and will be the destination for data written to the socket with functions other than standard functions `sendto()` and `sendmsg()`. The *lhost* argument presently is used only to provide the address of a multicast group if you want to receive multicast messages.

1.10 socket_client_max_retries

```
void socket_client_max_retries(struct socket_instance *sock,
                               int max_retries)
```

This function sets *sock*`->server_set.max_server_retries` to *max_retries*. This is the maximum number of times the library will try to connect to a server without success. Note that once a connection has been established the failure count is reset to zero, so subsequent connection attempts (after the connection breaks, for example) may be retried *max_retries* times.

1.11 socket_client_retry_interval

```
void socket_client_retry_interval(struct socket_instance *sock,
                                  float interval)
```

This function sets *sock*`->server_set.server_retry_interval` to *interval*. This is the number of seconds the library will wait after an unsuccessful connection attempt in `socket_tcp_connect()` (which also is called from `socket_setup()` for TCP client sockets).

1.12 socket_client_max_cycles

```
void socket_client_max_cycles(struct socket_instance *sock,
                              int max_cycles)
```

This function sets *sock*`->server_set.max_server_cycles` to *max_cycles*. This is the maximum number of times the library will cycle through the server list without making connection. Note that once a connection has been established the failure count is reset to zero, so subsequent connection attempts (after the connection breaks, for example) may cycle through the server list *max_cycles* times.

2 Higher-level functions

2.1 socket_setup

```
int socket_setup(struct socket_instance *sock,
                 int multicast_join_ok)
```

This is the simplest function in the library for creating a socket. Once *sock* is initialized and populated, a call to this function will return the file descriptor of a connected TCP or UDP socket, and the `local` and `foreign` elements of *sock* will be set.

The *multicast_join_ok* argument is used only for UDP sockets. If non-zero and the given address is a multicast group, the application will join the multicast group and be able to receive multicast messages. Also, if *multicast_join_ok* is non-zero and either a local host or port were given, the `SO_REUSEADDR` option is set on the socket. This is intended to allow multiple applications to receive multicast or broadcast data on the same port on the same machine, but it is applied to unicast addresses also. *Beware!*

The function returns -1 on errors.

2.2 socket_initiate

```
int socket_initiate(struct socket_instance *sock,
                    int *result, int multicast_join_ok)
```

Forming TCP socket connections can be thought of as a two-step process, whether the application is client or server. As a server, the application first creates a listen socket, and then accepts a connetion on that socket. As a client, the application can request a connection, and then discover the outcome of the request.

This function performs the first step in establishing a connection: it listens if *sock* is to be a server-side socket, and it requests a connection if it is to be s client-side socket. It returns a file descriptor which can be used in a call to `select()` to determine when it is time to call `socket_complete()` to finish setting up the connection. The returned descriptor should be checked for reading and writing by `select()`.

For UDP sockets, `socket_initialize()` returns a usable socket descriptor, but the application can treat a UDP socket instance the same as TCP instance – pass the returned descriptor to `select()` and subsequently call `socket_complete()` – and get correct results.

The function returns -1 on errors. Additionally, the *result* argument is set to -1 on errors, 0 if the connection is established, and 1 if the connection is pending.

2.3 socket_complete

```
int socket_complete(struct socket_instance *sock)
```

This function is the complement of `socket_initiate()`: It performs the second of the two steps in forming a TCP connection. If *sock* is a server-side socket, the function accepts a connection request (subject to TCP Wrappers approval); and, if *sock* is a client-side socket, the function determines the outcome of the connection request made by `socket_initiate()`.

The function returns a file descriptor for the connected socket or -1 on errors.

2.4 socket_tcp_connect

```
int socket_tcp_connect(struct socket_instance *sock)
```

This function forms a TCP connection from the client side. It is called for TCP client sockets by `socket_setup()`, but also may be called directly. It will perform retries and failovers, if necessary, before returning.

The function returns the file descriptor for the connected socket or -1 on errors.

2.5 socket_close

```
void socket_close(struct socket_instance *sock,
                  int multicast_join_ok)
```

The function returns -1 on errors.

3 Lower-level functions

To take the active (client) role in setting up a TCP connection, first initialize a `socket_instance` structure using `socket_client_instance()`, `socket_client_failover`, and related functions. Then call `socket_set_foreign_server()` and then `socket_tcp_connect_initiate()`.

To take the passive (server) role in setting up a TCP connection, first initialize a `socket_instance` structure using `socket_server_instance()`. Then call `socket_tcp_listen()` to create the listen socket.

To take any role using a UDP socket, initialize a `socket_instance` structure using `socket_udp_instance()` and call `socket_udp_create()`. This function handles multicast and broadcast sockets as well as unicast.

3.1 socket_tcp_connect_initiate

This function will initiate a non-blocking socket connection; that is, it will call `connect()` and return the file descriptor on which it called `connect()`.

The function's second argument, *result*, is used to return the status of the pending connection. If the value of *result* is zero, the connection has been established and the returned descriptor can be used immediately.

If the value of *result* is greater than zero, the connection is in progress. In this case, the returned file descriptor can be used in a call to `select()`. Wait for the descriptor to be readable *or* writable; that is, put the descriptor in both the read and write file descriptor sets for `select()`. When select indicates that the descriptor is ready, call `socket_tcp_connect_complete()` to finish the job.

BEWARE: On Solaris systems, `select()` will return -1 and set `errno` to `ENOSYS` if the connect() ultimately fails and the socket descriptor is in the write descriptor set!

3.2 socket_tcp_connect_complete

This function completes the operation began with `socket_tcp_connect_initiate()`. It returns the file descriptor for the connected socket or -1 if the connection attempt failed. The global errno should be set to indicate the error.

3.3 socket_set_foreign_server

Call this function in preparation for calling `socket_tcp_connect_initiate()` for the first time, or to attempt a retry or failover. If `socket_set_foreign_server()` return zero, you can call `socket_tcp_connect_initiate()`. If it returns non-zero, you have exhausted the retries and failovers programmed into the `socket_instance` given by *sock*.

3.4 socket_tcp_listen

This function prepares up the passive (or server) side of a TCP connection; that is, it will call `listen()` and return the descriptor on which it is listening. Note that the descriptor returned is set to non-blocking!

To accept a connection request, call `socket_tcp_accept()`.

3.5 socket_tcp_accept

This function completes a TCP connection from the passive (or server) side; that is, it calls `accept()` and returns the descriptor of the connected socket. If `continue_listening` is set in the `socket_instance`, the listen socket will remain open and you can make further calls to this function to accept additional connections; else, the listen socket will be closed.

3.6 socket_udp_create

This function create a socket for transmitting and/or receiving UDP datagrams. If `socket_udp_instance()` was called with a foreign host and port, data written to the returned descriptor will be sent to that peer socket; else, data may not be written to the returned descriptor.

3.7 socket_data_fd

Returns the file descriptor of the socket associated with this `socket_instance` on which data may be transferred.

3.8 socket_listen_fd

Returns the file descriptor of the socket associated with this `socket_instance` on which we are listening for a TCP connection request. Such a descriptor is associated only with server-side (or "passive") sockets.

Index to functions

(Index is nonexistent)

Table of Contents